CACHE-OBLIVIOUS SELECTION IN SORTED $X + Y$ MATRICES

MARK DE BERG AND SHRIPAD THITE

Abstract. Let $X[0..n-1]$ and $Y[0..m-1]$ be two sorted arrays, and define the $m \times n$ matrix A by $A[j][i] = X[i] + Y[j]$. Frederickson and Johnson [7] gave an efficient algorithm for selecting the kth smallest element from A. We show how to make this algorithm IO-efficient. Our cache-oblivious algorithm performs $O((m+n)/B)$ IOs, where B is the block size of memory transfers.

1. Introduction

Let S be a multi-set of elements from a totally ordered universe and let k be an integer in the range $1 \leqslant k \leqslant |S|$. The *selection problem* is to find a kth smallest element of S, that is, an element $x \in S$ that is kth in some non-decreasing total ordering of S. Selection is a fundamental problem in computer science and a key building block of many algorithms. Selection is trivial when S is sorted, but when S is not given in sorted order it becomes more challenging. A classical divide-and-conquer algorithm [4, 5] solves the selection problem for unsorted inputs in $O(|S|)$ time.

Often, the input is naturally organized as a two-dimensional matrix A with m rows and n columns. Using the classical algorithm one can perform selection in A in $O(mn)$ time, which is optimal in the worst case. When the rows and columns of the matrix are sorted, however, one can do much better. Frederickson and Johnson [7, 8] gave an algorithm for this case—we will call it the *FJ-algorithm* from now on—that runs in $O(m \lg(2n/m))$ time; here we assume without loss of generality that $m \leqslant n$. Note that when $m = n$ the running time is simply $O(n)$.

In some applications the matrix A is defined succinctly by the Cartesian product of two given vectors $X[0..n-1]$ and $Y[0..m-1]$. We are interested in the case where $A = X + Y$, that is,

$$A[j][i] = X[i] + Y[j]$$

where X and Y are sorted. (The symbol '+' can mean any monotone binary operator.) Since X and Y are sorted, the rows and columns of A are sorted. Hence, one can perform selection in A in $O(m \lg(2n/m))$ time by FJ-algorithm. Selection in such sorted $X + Y$ matrices is used as a subroutine in several other algorithms—see [1, 3, 6, 10, 13, 12] for some examples.

The FJ-algorithm is efficient in terms of CPU computation time. Unfortunately, it is not efficient when it comes to IO behavior, because it accesses elements of the input arrays X and Y non-sequentially, in a pattern that does not exhibit locality of reference. This is the goal of our paper: to develop a variant of the algorithm that has better IO behavior.

The *input-output complexity*, or *IO-complexity*, of an algorithm is usually analyzed in the external-memory model introduced by Aggarwal and Vitter [2]. In this model the memory consists of two levels: a fast memory and a slow memory. The fast memory can store up to M words and the slow memory has unlimited storage capacity. Data is stored in the slow memory in blocks of size B. To be able to do computations on data in the slow memory, that data first has to be brought into the fast memory; data which is evicted from fast memory (to make room for other data) needs to be written back to the slow memory. Data is transferred between fast and slow memory in blocks. The IO-complexity of an algorithm is the number of block transfers it performs.

The two levels in this abstract model can stand for any two consecutive levels in a multi-level memory hierarchy: the slow memory could be the disk and the fast memory the main memory, the slow memory could be the main memory and the fast memory the L3 cache, and so on. The values

of M and B are different at different levels; the higher up in the memory hierarchy, the larger the memory size M and block size B.

Our main result is a variant of the FJ-algorithm for sorted $X+Y$ matrices whose IO-complexity is $O(\textsc{Scan}(n+m))$. Here, $\textsc{Scan}(s)$ is the number of IOs performed when scanning s consecutive items; $\textsc{Scan}(s) \leqslant 1 + \lceil s/B \rceil$. Our algorithm is *cache-oblivious* [9], which means it is oblivious of the parameters M and B. In other words, the parameters M and B are only used in the analysis of the algorithm; they are not used in the algorithm itself. The beauty of cache-oblivious algorithms is that, since they do not depend on the values M and B, they are IO-efficient for all values of M and B and, hence, IO-efficient at all levels of a multi-level memory hierarchy.[1]

2. The FJ-Algorithm

First, we give a rough outline of the FJ-algorithm [7]. A detailed description is given in Figure 1.

Let $X[0..n-1]$ and $Y[0..m-1]$ be two input arrays of real numbers, given in sorted order: $X[0] \leqslant X[1] \leqslant \cdots \leqslant X[n-1]$ and $Y[0] \leqslant Y[1] \leqslant \cdots \leqslant Y[m-1]$. Let $A[0..m-1][0..n-1]$ be the matrix $X+Y$, that is, the matrix defined by $A[j][i] = X[i] + Y[j]$. We assume that $m = n$ and that n is a power of 2; this can easily be ensured by implicitly padding the arrays X and Y suitably.

Following Frederickson and Johnson, we call a submatrix of A a *cell*. The algorithm maintains a set $\mathcal{C}$ of *active cells*, such that the desired element will be present in one of the active cells. Initially, the entire matrix A is the sole active cell.

The algorithm proceeds in $\lg n$ iterations. Let $\mathcal{C}_p$ denote the set of active cells at the beginning of the pth iteration, where $p = 1,2,\ldots,\lg n$. The pth iteration begins by splitting each cell of $\mathcal{C}_p$ into four smaller cells by bisecting each dimension. Let $\mathcal{C}_p^*$ denote the list of cells obtained by splitting each cell of $\mathcal{C}_p$ into four. The algorithm next discards certain cells from $\mathcal{C}_p^*$ which do not contain the desired element, thus obtaining the set $\mathcal{C}_{p+1}$ to be used in the next iteration.

Cells are discarded based on their minimum and maximum elements. A cell $C \in \mathcal{C}_p^*$ for which $\min(C)$ is larger than a certain number of other minima can safely be discarded because all elements of C will be larger than the desired element. Similarly, a cell $C \in \mathcal{C}_p^*$ for which $\max(C)$ is smaller than a certain number of other maxima can be discarded because all elements of C will be smaller than the desired element. The exact condition for discarding cells is given in step (2b) of the algorithm in Figure 1.

The cells in $\mathcal{C}_p$ have size $(n/2^{p-1}) \times (n/2^{p-1})$ and the cells in $\mathcal{C}_p^*$ have size $(n/2^p) \times (n/2^p)$. Hence, after iteration $p = \lg n$, the cells in $\mathcal{C}_p$ are singletons (that is, 1×1 cells). The classical selection algorithm is then used to find the desired element among these singletons.

The following theorem stating the performance of the FJ-algorithm is a special case of the general theorem proved by Frederickson and Johnson [7].

Theorem 1. [7] *Given two sorted arrays X and Y, each of size n, the FJ-algorithm correctly computes an element of rank k in the matrix $A = X + Y$ in $O(n)$ time.*

3. IO-Efficient Selection

Next, we show how to make the algorithm of the previous section IO-efficient. Henceforth, we will refer to the slow memory in our two-level hierarchy as the *disk* and to the fast memory as the *cache*. We assume that the array X is laid out in order in n consecutive memory locations on disk. Similarly, the array Y is laid out in order in n consecutive memory locations on disk.

The FJ-algorithm needs an efficient selection algorithm in steps (2b), (2c), and (3). Fortunately, the standard selection algorithm has good IO-behavior.

[1] In the analysis of cache-oblivious algorithms it is assumed that the operating system uses an optimal block replacement strategy—see the paper by Frigo *et al.* [9] for a justification of this and some other assumptions in the model.

FJ-ALGORITHM(X, Y, k):

(1) Initialize $\mathcal{C}_1$ such that its only cell is the entire matrix $A = X + Y$.
(2) **for** $p := 1$ **to** $\lg n$ **do**
  (a) Split each $C \in \mathcal{C}_p$ into four subcells to obtain the set $\mathcal{C}_p^*$. Let $L_p := \min\{n, 2^{p+1} - 1\}$.
  (b) Let $q := \lceil k4^p/n^2 \rceil + L_p$.
    if $q \leqslant |\mathcal{C}_p^*|$
    then Use a standard selection algorithm to select a qth element x_u in the multiset $\{\min(C) : C \in \mathcal{C}_p^*\}$. Discard $|\mathcal{C}_p^*| - q + 1$ cells from $\mathcal{C}_p^*$, retaining every cell C with $\min(C) < x_u$ and no cell with $\min(C) > x_u$.
  (c) Let $r := \lceil k4^p/n^2 \rceil - L_p$.
    if $r \geqslant 1$
    then Use a standard selection algorithm to select an rth element x_l in the multiset $\{\max(C) : C \in \mathcal{C}_p^*\}$. Discard r cells from $\mathcal{C}_p^*$, retaining every cell C with $\max(C) < x_l$ and no cell with $\max(C) > x_l$.
  (d) Let $k := k - r(n^2/4^p)$ and let $\mathcal{C}_{p+1} := \mathcal{C}_p^*$.
(3) Select the kth element from the cells in $\mathcal{C}_p$ using a standard selection algorithm.

FIGURE 1. The matrix selection algorithm of Frederickson and Johnson [7].

Lemma 2. *The standard selection algorithm* [4] *selects an element of a given rank k from an array of s elements in $O(s)$ time and using $O(\text{SCAN}(s))$ IOs.*

Even though selection is the main subroutine used by the matrix selection algorithm, Lemma 2 does not imply that the FJ-algorithm is IO-efficient. The main problem is that maintaining the list of active cells can dominate the IO-cost of a naïve implementation of the FJ-algorithm, leading to $O(n)$ IO-complexity rather than $O(\text{SCAN}(n))$. To make the algorithm IO-efficient, we need to take a detailed look at the manipulation of active cells.

The FJ-algorithm needs a data structure to store the sets $\mathcal{C}_p$ and $\mathcal{C}_p^*$ of active cells. One could use linked lists, but traversing a linked list is not IO-efficient because adjacent list elements could be stored in different blocks, requiring as many as one IO-operation per list element. Instead, we use arrays, which can store any list L compactly on disk in $O(|L|/B)$ blocks.

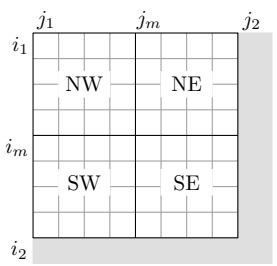
We represent a cell $A[j_1..j_2 - 1][i_1..i_2 - 1]$ by the 8-tuple

$$(i_1, j_1, i_2, j_2, X[i_1], X[i_2 - 1], Y[j_1], Y[j_2 - 1]),$$

and we identify a cell with its corresponding 8-tuple. The active cells are stored in lexicographic order of their corresponding 8-tuples. From the 8-tuple representing cell C we can compute $\min(C) = X[i_1] + Y[j_1]$ and $\max(C) = X[i_2 - 1] + Y[j_2 - 1]$ in $O(1)$ time and no additional IOs. Hence, steps (2b), (2c), and (3) of the FJ-algorithm can all be done in $O(\text{SCAN}(|\mathcal{C}_p^*|))$ IOs. The problem lies in step (2a), where we compute $\mathcal{C}_p^*$ from $\mathcal{C}_p$ by splitting each cell into four subcells.

Suppose we have to split the cell $(i_1, j_1, i_2, j_2, X[i_1], X[i_2], Y[j_1], Y[j_2])$. Let $i_m = (i_1 + i_2)/2$ and let $j_m = (j_1 + j_2)/2$. The four subcells we must generate are as follows:

north-west: $(i_1, j_1, i_m, j_m, X[i_1], X[i_m - 1], Y[j_1], Y[j_m - 1])$
north-east: $(i_1, j_m, i_m, j_2, X[i_1], X[i_m - 1], Y[j_m], Y[j_2 - 1])$
south-west: $(i_m, j_1, i_2, j_m, X[i_m], X[i_2 - 1], Y[j_1], Y[j_m - 1])$
south-east: $(i_m, j_m, i_2, j_2, X[i_m], X[i_2 - 1], Y[j_m], Y[j_2 - 1])$



Most components of the subcells can be computed from the components of C, except that $X[i_m - 1]$

and $X[i_m]$ need to be fetched from the array X, and $Y[j_m - 1]$ and $Y[j_m]$ need to be fetched from the array Y. If we are not careful, fetching these values will cost us an IO each time and the whole algorithm will not be IO-efficient. Next we describe how to overcome this problem.

Let us examine in what order the algorithm accesses the array X; the array Y will be discussed later.

First consider the elements from X needed for the fifth component of the cells, which stores the minimum X-value in the cell. In the initialization step, the entire matrix A is the only active cell; its minimum X-value is $X[0]$. In the first iteration ($p = 1$) we split A into four subcells. The minimum X-values in those subcells are either $X[0]$ (for the north-west and south-west subcells) or $X[n/2]$ (for the north-east and south-east subcells). Since $X[0]$ is conveniently stored in the original cell, we only need to access $X[n/2]$. In the second iteration, we need to access $X[n/4]$ and $X[3n/4]$. In general, in the pth iteration ($1 \leqslant p < \lg n$), each active cell in $\mathcal{C}_p$ has dimension $(n/2^{p-1}) \times (n/2^{p-1})$, and the elements that need to be accessed to obtain their minimum X-values are $X[(2i-1) \cdot (n/2^p)]$ for $1 \leqslant i \leqslant 2^{p-1}$. (In fact, we do not necessarily need all these elements, since not all cells have to be active.)

To enable IO-efficient access to these elements in X, we construct an array $X_1[1..n/2 - 1]$ that stores, for any p with $1 \leqslant p < \lg n$, the elements needed in the pth iteration consecutively. Thus we define array X_1 so that it has the following property:

For all p in the range $1 \leqslant p < \lg n$, for all i in the range $1 \leqslant i \leqslant 2^{p-1}$, we have

$$X_1\left[2^{p-1} + i - 1\right] = X\left[(2i-1)\frac{n}{2^p}\right]. \tag{1}$$

Note that, together with $X[0]$, the elements in X_1 are exactly the elements in X at even-numbered positions.

Similarly, the elements that need to be accessed to obtain the maximum X-values in the p-th iteration, namely $X[(2i-1) \cdot (n/2^p - 1)]$ for $1 \leqslant i \leqslant 2^{p-1}$, are stored in an array X_2. Thus array $X_2[1..n/2 - 1]$ stores the odd-numbered elements in X (except $X[n-1]$), as follows:

For all p in the range $1 \leqslant p < \lg n$, for all i in the range $1 \leqslant i \leqslant 2^{p-1}$, we have

$$X_2\left[2^{p-1} + i - 1\right] = X\left[(2i-1)\,\frac{n}{2^p} - 1\right]. \tag{2}$$

Next we show how to compute the array X_1 efficiently; X_2 can be computed similarly.

Given an integer i, the *bit-reversal* of i is the integer $\beta(i)$ such that the binary string representing $\beta(i)$ is the reverse of the binary string representing i. The *bit-reversal permutation* Z' of an array Z is the permutation that maps that $Z[i]$ to $Z'[\beta(i)]$. The bit-reversal permutation can be computed recursively as follows: Copy all elements in even-numbered positions in Z in order to the first half of the array Z', and copy all elements in odd-numbered positions of Z in order to the second half of Z'; recurse on both halves.

Now suppose we only recurse on the first half of the array Z'; the elements in the second half are kept in the same relative order as in the input array Z. We call the resulting permutation the *partial bit reversal*. As we will show below, the partial bit reversal of array X is closely related to the array X_1 that we want to compute. The recursive algorithm PBR given in Fig. 2—a non-recursive version would also be possible—computes a partial bit reversal Z' of a given array $Z[0..n-1]$. In the initial call, Z' is a copy of Z, and $s = n$. (Recall that we assumed n is a power of 2.)

The following lemma, which gives the running time and IO complexity of PBR, follows easily from the fact that steps 2a and 2b of PBR are just linear scans of arrays Z', *Even*, and *Odd*, so these steps run in $O(s)$ time and $O(\textsc{Scan}(s))$ IOs.

Lemma 3. *PBR*(Z', n) *runs in* $O(n)$ *time and uses* $O(\textsc{Scan}(n))$ *IOs.*

ALGORITHM PBR(Z', s):

(1) **if** $s > 1$
(2) **then** *Comment: Even*$[0..s/2-1]$ *and Odd*$[0..s/2-1]$ *are auxiliary arrays.*
  (a) **for** $i := 0$ **to** $s-1$ **do**
    if i is even **then** $Even[i/2] := Z'[i]$ **else** $Odd[(i-1)/2] := Z'[i]$
  (b) **for** $i := 0$ **to** $s/2-1$ **do** $Z'[i] := Even[i]$
    for $i := s/2$ **to** $s-1$ **do** $Z'[i] := Odd[i - s/2]$
  (c) PBR($Z', s/2$)

FIGURE 2. Algorithm to compute a partial bit-reversal permutation

The next lemma shows the correspondence between the partial bit reversal of our input array X and the array X_1 we want to compute. It implies that X_1 can be obtained by computing the partial bit reversal X' of X and then taking the elements from $X'[1..n/2-1]$ in order.

Lemma 4. *Let Z' be the partial bit reversal of an array $Z[0..n-1]$, where n is a power of 2, as computed by PBR. Then for all p in the range $1 \leqslant p < \lg n$, for all i in the range $1 \leqslant i \leqslant 2^{p-1}$, we have*

$$Z'\left[2^{p-1} + i - 1\right] = Z\left[(2i-1)\frac{n}{2^p}\right].$$

Proof. Let $j > 0$ be an even index, and let $\ell \geqslant 1$ and $k \geqslant 1$ be such that $j = (2\ell - 1)2^k$. Now consider what happens to element $Z[j]$. Initially Z' is a copy of Z, so $Z[j]$ is stored in $Z'[j]$. Then, in the first call to PBR—that is, the call with $s = n$—it will be moved to $Z'[j/2]$ by steps (2a) and 2b. In the recursive call with $s = n/2$ it will be moved to $Z'[j/4]$ (if $k > 1$). This process continues k times, until the recursive call is made with $s = n/2^k$. At this point $Z[j]$ is stored in $Z'[2\ell - 1]$, and step (2a) moves the element to $Z'[n/2^{k+1} + \ell - 1]$. After that it will not be moved anymore by the algorithm.

Now set $i = \ell$ and take p such that $2^k = n/2^p$. Then $n/2^{k+1} = 2^{p-1}$ and we can conclude that $Z[(2i-1)\cdot(n/2^p)]$ ends up in $Z'[2^{p-1} + i - 1]$, as required. □

In what follows, we use $\beta_1(j)$ to denote the position of $X[j]$ in the array X_1, for $j > 0$ and j even. Thus, according to Equation (1), we have $\beta_1((2i-1)\cdot(n/2^p)) = 2^{p-1}+i-1$. Similarly, $\beta_2(j)$ denotes the position of $X[j]$ in the array X_2, for $j < n-1$ and j odd; thus $\beta_2((2i-1)\cdot(n/2^p)-1) = 2^{p-1}+i-1$.

The arrays X_1 and X_2 give us the $X[\cdot]$-values in the order they are needed by the cell-partitioning step of the FJ-algorithm. However, to partition a cell we also need to fetch new $Y[\cdot]$ values. For this we would like to use the same approach: compute in a preprocessing step two arrays $Y_1[1..n/2-1]$ and $Y_2[1..n/2-1]$, which contain the $Y[\cdot]$-values in the order needed by the algorithm. With the $X[\cdot]$-values this approach was possible, because the cells in $\mathcal{C}_p$ are kept in lexicographical order, with the i_1-value being dominant. Hence, we knew exactly not only which $X[\cdot]$-values were needed in the p-th iteration (namely $X[(2i-1)\cdot(n/2^p)]$ for $1 \leqslant i \leqslant 2^{p-1}$), but also in which order (namely according to increasing index). But for the $Y[\cdot]$-values we only know which values we need in the p-th iteration; we do not know in which order we need them, because the i_1-coordinate is dominant in the order of the cells in $\mathcal{C}_p$. Next we will show that the approach works nevertheless. Thus we compute arrays Y_1 and Y_2 in exactly the same way as the arrays X_1 and X_2 were computed. Then we partition the cells with the algorithm shown in Figure 3.

Before we can prove that this algorithm is indeed IO-efficient, we need to deal with one subtlety: we need to be more specific about the exact implementation of step (2b) of the FJ-algorithm in case the qth element, x_u, is not unique. More precisely, we need to specify which of the cells C

PARTITION$(\mathcal{C}_p, X_1, X_2, Y_1, Y_2, p)$:

(1) Let $\mathcal{C}_{p,R}$ and $\mathcal{C}_{p,L}$ be two arrays of twice the size as $\mathcal{C}_p$.
(2) **for** $i := 0$ **to** $|\mathcal{C}_P| - 1$ **do**
  Let $C = (i_1, j_1, i_2, j_2, X[i_1], X[i_2 - 1], Y[j_1], Y[j_2 - 1])$ be the cell in $\mathcal{C}_p[i]$.
  Let $i_m = (i_1 + i_2)/2$ and let $j_m = (j_1 + j_2)/2$.
    (a) Fetch $X[i_m - 1]$ from $X_1[\beta_1(i_m - 1)]$ and $X[i_m]$ from $X_2[\beta_2(i_m)]$
    (b) Fetch $Y[j_m - 1]$ from $Y_1[\beta_1(j_m - 1)]$ and $Y[j_m]$ from $Y_2[\beta_2(j_m)]$
    (c) $\mathcal{C}_{p,L}[2i] \leftarrow (i_1, j_1, i_m, j_m, X[i_1], X[i_m - 1], Y[j_1], Y[j_m - 1])$
    (d) $\mathcal{C}_{p,L}[2i + 1] \leftarrow (i_m, j_1, i_2, j_m, X[i_m], X[i_2 - 1], Y[j_1], Y[j_m - 1])$
    (e) $\mathcal{C}_{p,R}[2i] \leftarrow (i_1, j_m, i_m, j_2, X[i_1], X[i_m - 1], Y[j_m], Y[j_2 - 1])$
    (f) $\mathcal{C}_{p,R}[2i + 1] \leftarrow (i_m, j_m, i_2, j_2, X[i_m], X[i_2 - 1], Y[j_m], Y[j_2 - 1])$
(3) *Comment: Now $\mathcal{C}_{p,R}$ and $\mathcal{C}_{p,L}$ together contain the new subcells, and both arrays are sorted lexicographically.*
(4) Merge $\mathcal{C}_{p,R}$ and $\mathcal{C}_{p,L}$ into an array $\mathcal{C}_p^*$ that is sorted lexicographically.
(5) **return** $\mathcal{C}_p^*$.

FIGURE 3. Partitioning each cell in $\mathcal{C}_p$ into four subcells.

with $\min(C) = x_u$ are discarded and which are kept. Similarly, we must specify which of the cells C with $\max(C) = x_l$ are discarded and which are kept in step (2c). We do this as follows.

Recall that we maintain $\mathcal{C}_p^*$ in lexicographic order. Now we can implement step (2b) by removing from $\mathcal{C}_p^*$ exactly those cells whose ranks are greater than q according to this lexicographical order. This implies that if we remove a certain cell C, we will also remove all cells to the south-east of C (including the ones to the south of C, and the ones to the east of C). We use a similar strategy to guarantee that when we remove a cell in step (2c), we also remove all cells to its north-west. With this implementation, the active cells have the following properties—see also Figure 3.

(i) All active cells with the same column index are consecutive.
(ii) The active cell with the largest row index in a given column—note that row indices increase when going downwards in Figure 3—cannot have row index smaller than the any active cell in the column to its right. In other words, if we consider the lowest active cells in each column and we consider the columns from left to right, then the the row indices of these highest active cells are non-increasing.

These properties are essential to get good IO-complexity of PARTITION.

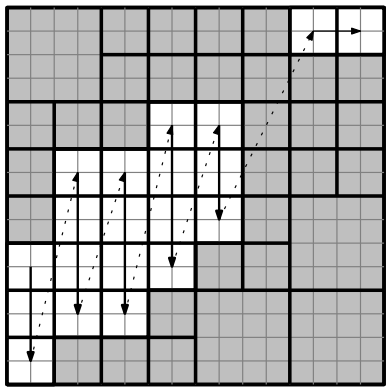


FIGURE 4. The structure of the active cells, and the order in which they are accessed (which is the lexicographic order). Active cells are white, discarded cells are grey.

Lemma 5. *Algorithm* PARTITION *produces a lexicographically sorted array $\mathcal{C}_p^*$ of all subcells resulting from partitioning every cell in $\mathcal{C}_p$ into four.* PARTITION *runs in $O(|\mathcal{C}_p|)$ time and performs $O(\text{SCAN}(|\mathcal{C}_p| + 2^p))$ IOs.*

Proof. The correctness of the algorithm directly follows from the fact that, by definition of β_1 and β_2, the correct values are fetched in steps (2a) and (2b).

To bound the running time, we note that $\beta_1(\cdot)$ and $\beta_2(\cdot)$ can be evaluated in $O(1)$ time. Indeed, when we evaluate e.g. $\beta_1(j)$ for some j, we know the value of p such that $j = (2i-1) \cdot (n/2^p)$—this p is a parameter of PARTITION. Given p, we have $\beta_1(j) = 2^{p-1} + (j \cdot (2^p/n) + 1)/2 - 1$. It follows that the running time is $O(n)$.

As for the number of IOs, all accesses to $\mathcal{C}_p$, as well as step (4), take $O(\text{SCAN}(|\mathcal{C}_p| + 2^p))$ IOs in total. Hence, it remains to argue about the accesses to X_1, X_2, Y_1, and Y_2.

We first consider the accesses to X_1. As argued earlier, the cells in $\mathcal{C}_p$ have size $(n/2^{p-1}) \times (n/2^{p-1})$, which means we need to fetch from X_1 (a subset of) the elements $X[(2i-1) \cdot (n/2^{p-1})]$ for $1 \leqslant i \leqslant 2^{p-1}$. By the definition of X_1—see Equation (1)—these elements are consecutive in X_1. Moreover, these elements are accessed from left to right in X_1, because the cells in $\mathcal{C}_p$ are sorted in increasing order of their first coordinate. Hence, all these accesses to X_1 take $O(\text{SCAN}(2^{p-1})) = O(\text{SCAN}(2^p))$ IOs in total. Symmetric reasoning gives the same bound on the number of accesses to X_2.

Now consider the accesses to Y_1; symmetric reasoning bounds the accesses to Y_2. Consider Figure 3. The active cells will be visited by the algorithm in lexicographic order, as indicated in the figure. This means that the algorithm may go back and forth in Y_1. Moreover, when going back, the algorithm may *jump* from accessing some element $Y_1[j]$ to accessing another element $Y_1[j']$ where $j - j' > 1$; we call $j - j'$ the *length* of the jump. Jumps are significant because each jump may incur a cost of one IO operation. (Jumps are also possible when accessing X_1 or X_2. Since in X_1 and X_2 we only jump forward, this does not increase the number of IOs there.) Note that the elements needed within a single column of active cells, are stored in the correct order in Y_1. (Here the term "column" refers to a column in the matrix of whose cells are submatrices of size $(n/2^p) \times (n/2^p)$.) When we step from the lowest active cell in one column to highest active cell in the next column, however, we may jump in Y_1. Now suppose that instead of jumping from one location to the next, we visit all intermediate locations as well. Hence, after visiting $Y_1[j]$, the new traversal always proceeds to either $Y_1[j-1]$ or $Y_1[j+1]$. We call such a traversal *well-behaved*. Clearly the number of IOs needed by the new traversal of Y_1 is not more than the number of traversals needed by the original traversal.

The original traversal visited $|\mathcal{C}_p|$ (not necessarily distinct) locations in Y_1. We claim that the length of the new, well-behaved traversal is $O(|\mathcal{C}^p| + 2^p)$. To show this, we must bound the total length of all backward jumps. Consider a backward jump from the lowest active cell in some column C to the highest active cell in the next column C'. This jump crosses a number of rows. By properties (i) and (ii) of the active cells, for each row that is crossed, at least one of the following three condition holds: C contains an active cell in this row, C' contains an active cell in this row, or the row will not be visited again later. This is easily seen to imply that the total length of all jumps is $O(|\mathcal{C}^p| + 2^p)$, as claimed.

It remains to observe that, assuming $M \geqslant 2B$—that is, assuming at least two blocks fit in the cache—any well-behaved traversal of length L needs $\text{SCAN}(L)$ IOs. Indeed, suppose we need to read a new block when we step from $Y_1[i]$ to $Y_1[i+1]$. Then we read the block starting at $Y_1[i+1]$ and can keep the block ending at $Y_1[i]$ in cache. Hence, at least $B - 1$ more forward steps or at least B backward steps are needed before another block needs to be read. We conclude that the number of IOs performed in accessing Y_1 (and, similarly, Y_2) is $O(\text{SCAN}(|\mathcal{C}^p| + 2^p)$, which finishes the proof for the number of IOs. □

Theorem 6. *There exists a cache-oblivious implementation of the matrix selection algorithm of Frederickson and Johnson for sorted $X + Y$ matrices using $O(\text{SCAN}(n))$ IOs and $O(n)$ time, where n is the maximum of the lengths of X and Y.*

Proof. By Lemma 3, the computation of the arrays X_1, X_2, Y_1, and Y_2 takes $O(\text{SCAN}(n))$ IOs and $O(n)$ time. Now consider the main algorithm. Frederickson and Johnson [7] proved that $|\mathcal{C}^p|$, the number of active cells in the beginning of the pth iteration, is $O(2^p)$. By Lemmas 2, 4, and 5, this implies that the total IO-cost is bounded by

$$\sum_{p=1}^{\lg n} O(\text{SCAN}(2^p)) = O(\text{SCAN}(n)).$$

Since the subroutine PARTITION runs in $O(|\mathcal{C}_p|)$, the running time of the main algorithm is unchanged from the original FJ-algorithm, which runs in $O(n)$ time. □

4. Conclusion

In this paper, we gave an IO-efficient cache-oblivious version of the classical matrix selection algorithm of Frederickson and Johnson for selecting a rank-k element in an $n \times n$ matrix given succinctly in the form $A := X + Y$.

If the matrix A is not square—that is, if its dimensions were $m \times n$ where $m < n$—then a different approach seems to be required to make the matrix selection algorithm IO-efficient. One would like to obtain an IO-cost of

$$O\left(\frac{m}{B}\log_B \frac{2n}{m}\right).$$

However, we already spend $O((m+n)/B)$ IOs in permuting the input arrays as a pre-processing step, which dominates the IO-cost of the subsequent algorithm. It seems difficult to avoid the high IO-cost of permuting both input arrays so that they can be accessed IO-efficiently. A completely new algorithm may be necessary to achieve IO-optimal matrix selection in sorted $X + Y$ matrices that are not square.

References

[1] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Computing Surveys*, 30(4):412–458, 1998.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] M. de Berg, O. Devillers, M. van Kreveld, O. Schwarzkopf, and M. Teillaud. Computing the maximum overlap of two convex polygons under translations. *Theory Comput. Syst.* 31:613–628, 1998.

[4] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Comput. System Sci.*, 7:448–461, 1973.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd ed. edition, 2001.

[6] A. Efrat and M.J. Katz. Computing fair and bottleneck matchings in geometric graphs. In *Proc. 7th Int. Symposium on Algorithms and Computation (ISAAC)*, pages 115–125, 1996.

[7] G. N. Frederickson and D. B. Johnson. Generalized selection and ranking: Sorted matrices. *SIAM J. Computing*, 13:14–30, 1984.

[8] G. N. Frederickson and D. B. Johnson. Erratum: Generalized selection and ranking: Sorted matrices. *SIAM J. Computing*, 19(1):205–206, 1990.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th IEEE Symp. Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[10] A. Golzman, K. Kedem, and G. Spitalnik. On some geometric selection and optimization problems via sorted matrices. In *Proc. 4th Workshop on Algorithms and Data Structures*, LNCS 955, pages 26–37, 1995.

[11] M.J. Katz and K.Kedem and M.Segal. Discrete rectilinear 2-center problems. *Comput. Geom. Theory Appl.* 15: 203–214, 2000.

[12] M. Sharir and E. Welzl. Rectilinear and polygonal p-piercing and p-center problems. In *Proc. 12th ACM Sympos. Comput. Geom.*, pages 122–132, 1996.

[13] T. Strijk, and M. van Kreveld. Labeling a rectilinear map more efficiently. *Inf. Proc. Lett.* 69(1):25–30, 1999.

Mark de Berg, Department of Computer Science, Technische Universiteit Eindhoven, The Netherlands

E-mail address: mdberg@win.tue.nl

Shripad Thite, California Institute of Technology, Center for the Mathematics of Information, USA

E-mail address: shripad@caltech.edu